Structural Resolution with Co-inductive Loop Detection

Yue Li
School of Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh, United Kingdom
yl55@hw.ac.uk

A way to combine co-SLD style loop detection with structural resolution was found and is introduced in this work, to extend structural resolution with co-induction. In particular, we present the operational semantics, called co-inductive structural resolution, of this novel combination and prove its soundness with respect to the greatest complete Herbrand model.

1 Introduction

Co-inductive logic programming extends traditional logic programming by enabling co-inductive reasoning to deal with infinite SLD-derivation, which has practical implication in different fields of computing such as model checking, planning as well as type inference [21, 20, 2, 15].

One operational semantics of co-inductive logic programming is co-inductive SLD resolution (co-SLD) [21, 20], which combines loop detection with traditional SLD resolution, so that it can be used to reason co-inductively about infinite rational terms. An alternative operational semantics for co-inductive logic programming is co-algebraic logic programming [15, 13], which adopts a more general co-induction rule and uses structural resolution instead of SLD resolution as its induction rule.

The distinctive features of co-algebraic logic programming, compared with co-SLD, include that the co-inductive reasoning mechanism of the former goes beyond loop detection; as a result the computed formulae by the former are allowed to be either rational terms or (finite observation of) irrational terms. Moreover, structural resolution [13, 11] allows for analysis of productivity [13] of logic programs. Productivity, also known as computations at infinity [17, ch. 4], concerns computation of infinite data structures by non-terminating derivations. Productivity is studied not only in logic programming community, but also in functional programming community, where they developed decision algorithm for co-recursive list and stream definitions [19, 8]. Those distinctive features make co-algebraic logic programming an ideal basis for developing productivity decision algorithms.

In this paper we explore a combination of co-SLD style loop detection with structural resolution, resulting in a novel operational semantics called *co-inductive structural resolution* (co-S-resolution for short), and we prove its soundness with respect to the greatest complete Herbrand model. An implementation is also presented as a contribution.

Co-inductive structural resolution is created as an intermediate semantics between co-SLD and co-algebraic logic programming. On the one hand, it inherits loop detection from co-SLD, which is a simpler co-inductive reasoning mechanism compared with the co-inductive mechanism of co-algebraic logic programming. On the other hand, it uses structural resolution as co-algebraic logic programming does, allowing for analysis of productivity. Introducing such an intermediate semantics has its practical implication: there are two challenges involved in the design of productivity decision algorithms, which are 1) productivity analysis with structural resolution and 2) co-inductive reasoning beyond loop detection; introducing the intermediate semantics makes it possible to deal with these *two* challenges *one at*

E. Komendantskaya and J. Power (Eds.):
Coalgebra, Horn Clause Logic Programming and Types
EPTCS 258, 2017, pp. 52–67, doi:10.4204/EPTCS.258.4

a time, so co-S-resolution prepares future development of a productivity decision algorithm that uses loop detection, which in turn will prepare even further algorithm development that goes beyond loop detection. These points will be further discussed in Section 4.

An overview of the rest of the paper is as follows. In Section 2 we will introduce preliminary concepts that cover substitution and unification with rational trees, co-SLD and structural resolution, and greatest fix point, which will prepare us for further theory construction in later sections. In Section 3 we will introduce the semantics for co-inductive structural resolution and prove its soundness. In Section 4 we will have a review of related work, discuss the importance of co-S-resolution for productivity decision, and conclude the paper. Appendix A presents the implementation.

2 Preliminaries

We assume readers' understanding of standard definition of *first order term* and the modelling of (possibly infinite) terms by *trees*. "Term" and "tree" are used interchangeably in this paper. Details about these concepts can be found in [17, 6].

Definition 1 (Rational Term)**.** A *rational term* [5, 10, 3, 21] refers to a (possibly non-ground and possibly infinite) term (or tree) that has a *finite* amount of distinct sub-terms (or sub-trees). A rational term is also known as a *regular term* [6, 9].

Our definition for *substitution with rational trees* (referred to as *substitution* for short hereinafter) inherits the principle of substitution with finite trees in logic programming [17].

Definition 2 (Substitution)**.** A substitution is a mapping of the form $S = \{x_1/t_1, \ldots, x_n/t_n\}$ where $x_1, \ldots, x_n$ are distinct variables, $t_1, \ldots, t_n$ are rational terms, and $\forall\, i, j \in \{1, \ldots, n\}$, x_i does not occur in t_j. Moreover, ε denotes the empty substitution.

Example 2.1. Let $\theta = \{x_1/f(f(\ldots)), x_2/g(x_3)\}$ be a substitution. Applying θ to the term $p(x_1, x_2)$ is denoted by $p(x_1, x_2)\theta$, which evaluates to $p(f(f(\ldots)), g(x_3))$.

Composition of substitutions is defined in the same way as in [17, Sec. 4].

Definition 3 (Unification)**.** Given two rational terms t_1 and t_2, unification is the process of finding a substitution (unifier) θ such that $t_1\theta = t_2\theta$, i.e. applying θ separately to t_1 and t_2 yields the same tree. This relation is denoted by $t_1 \sim_\theta t_2$.

The standard approach to rational term unification [18, 6, 5, 7, 23] involves *systems of equations of finite terms*, and *transforms* that turn equation systems to their *reduced form* as the output of the unification algorithm. The reduced form equation system can further be *solved* to obtain a solution in the domain of rational trees [6]. We omit the details of the unification algorithm for rational trees and the details of solving equation systems, which can be found in the above literature.

Remark. Our definition of substitution refers to solutions of reduced form equation systems. Consider the unification problem $p(X) \sim p(f(X))$. The standard approach regards $p(X) \sim p(f(X))$ as an equation system $\{p(X) = p(f(X))\}$ and reduces it to the reduced form $\{X = f(X)\}$, which is called a substitution in the standard sense. Locally in this paper we *solve* the reduced form $\{X = f(X)\}$ to obtain the solution $\{X = f(f(\ldots))\}$ and call this solution a substitution. We believe that our treatment of substitution can emphasize the variables that are to be instantiated and will make the theory about co-inductive structure resolution easier to formulate and understand.

Term matching is a concept closely related to unification, and is prerequisite for rewriting reduction in structural resolution [13, 11]. We extend applicable terms for matching from finite trees to rational trees by building the concept of rational tree term matching on the concept of rational tree unification.

Definition 4 (Term Matching)**.** Given two rational terms t_1, t_2 and a unifier σ, if σ also satisfies that $t_1\sigma = t_2$, then it is said that t_1 *subsumes* t_2, or t_1 *matches against* t_2. This relation is denoted by $t_1 \prec_\sigma t_2$. σ is called a *matcher* for t_1 against t_2.

Remark (Uniqueness of matcher). If two terms have matchers σ_1 and σ_2, then σ_1 equals to σ_2 when restricted to variables that occur in the terms. Nevertheless, a matcher σ for term t_1 against t_2 is intended to be identity on variables not in t_1.

The symbolic notation for unification ($\sim$) and matching ($\prec$) follows [13]. Note that ($\sim$) is a symmetric relation but ($\prec$) is not symmetric. Sometimes $t_1 \prec t_2$ may fail to convey which term subsumes (i.e. matches against) which. A mnemonic tip is to regard the "precede" symbol ($\prec$) as the "less than" symbol ($<$) and derive from $t_1 \prec t_2$ that t_2 might be bigger (i.e. contains more symbols) than t_1.

Example 4.1. a) $p(x) \sim_\theta p(f(x))$ where $\theta = \{x/f(f(f(\ldots)))\}$. θ is not a matcher.

b) $p(x_1,x_1) \sim_\theta p(f(y_1),y_1)$ where $\theta = \{x_1/f(f(f(\ldots))), y_1/f(f(f(\ldots)))\}$. θ is not a matcher.

c) $p(x_1,x_2) \sim_\theta p(f(y_1),y_1)$ where $\theta = \{x_1/f(y_1), x_2/y_1\}$. θ is a matcher and $p(x_1,x_2) \prec_\theta p(f(y_1),y_1)$.

We now introduce the operational semantics of the well-known SLD-resolution (**L**inear resolution for **D**efinite clauses with **S**election function) [17, 22] as a precursor of co-SLD and of structural resolution.

Definition 5 (SLD-resolution)**.** Given a logic program P and goal

$$G = \leftarrow A_1, \ldots, A_n$$

if there exists in P a clause $B_0 \leftarrow B_1, \ldots, B_m$ (with freshly renamed variables), such that $B_0 \sim_\theta A_k$ for some $k \in \{1, \ldots, n\}$, then by SLD-resolution we derive

$$G' = \leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta$$

Remark. The notation of the form $(A_1, \ldots, A_n)\theta$ denotes application of θ to every A_i, $i \in \{1, \ldots, n\}$.

In the following definition of co-SLD we introduce a set S for each predicate A in a goal [3], where S records all previous goals (or their instances) that are relevant to the co-inductive proof of A.

Definition 6 (co-SLD resolution)**.** Given a logic program P and a goal

$$G = \leftarrow (A_1, S_1), \ldots, (A_n, S_n)$$

the next goal G' can be derived by one of the following two rules:

1. If there exists in P a clause $B_0 \leftarrow B_1, \ldots, B_m$ (with freshly renamed variables), such that $B_0 \sim_\theta A_k$ for some $k \in \{1, \ldots, n\}$, then let $S' = S_k \cup \{A_k\}$, we derive

$$G' = \leftarrow \big((A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (B_1, S'), \ldots, (B_m, S'), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)\big)\theta$$

2. (Loop Detection) If $A_k \sim_\theta B$ for some $k \in \{1, \ldots, n\}$ and some $B \in S_k$, we derive

$$G' = \leftarrow \big((A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)\big)\theta$$

Remark. The notation of the form $\big((A_1, S_1), \ldots, (A_n, S_n)\big)\theta$ denotes application of θ to every A_i and to every member of every S_i, $i \in \{1, \ldots, n\}$.

Definition 7 (co-SLD Derivation/Refutation)**.** A co-SLD derivation consists of a possibly infinite sequence of goals $G_0, G_1, \ldots$ where G_0 is of the form $\leftarrow (A_1, \emptyset), \ldots, (A_m, \emptyset)$ $(m \geq 0)$, and for all $i \geq 0$, G_{i+1} is derived from G_i using co-SLD resolution. A finite co-SLD derivation ending with the empty goal is called an co-SLD refutation[1].

Definition 8 (Computed Substitution)**.** Given a co-SLD refutation $\mathscr{D}$, let $\theta_1, \theta_2, \ldots, \theta_n$ be the sequence of unifiers computed in $\mathscr{D}$ in the same order as they were computed, their composition $\theta_1\theta_2\cdots\theta_n$ is called the computed substitution from $\mathscr{D}$.

Co-SLD is co-inductively sound [3, 21]. Co-inductive soundness is defined in terms of the *greatest complete Herbrand model*. We assume the standard definition of complete Herbrand interpretation and complete Herbrand base [17, Sec. 25], which, compared with Herbrand interpretation/base, allow for infinite ground terms and atoms in addition to finite ones.

Definition 9 (T'_P operator)**.** Let P be a logic program and B'_P be P's complete Herbrand base. The complete immediate consequence operator $T'_P : 2^{B'_P} \mapsto 2^{B'_P}$ is defined as follows. Let $I \subseteq B'_P$ be a complete Herbrand interpretation. Then

$$T'_P(I) = \{A \in B'_P \mid A \leftarrow A_1, \ldots, A_n \text{ is a ground instance of a clause in P and } \{A_1, \ldots, A_n\} \subseteq I\}$$

Definition 10 (Greatest complete Herbrand model)**.** Let P be a program. The greatest fix point $\mathrm{gfp}(T'_P) = \cup\{I \mid I \subseteq T'_P(I)\}$ of T'_P is called the greatest complete Herbrand model of P.

More details on T'_P operator and its fix points can be found in e.g. [17, Sec. 26]. We now justify the loop detection rule of co-SLD with an example.

Example 10.1. Consider the following program P which defines co-recursively all streams of 0's and 1's.

bit(0)←
bit(1)←
bit-stream(cons(X,Xs)) ← bit(X), bit-stream(Xs)

The co-SLD refutation for goal ←bit-stream(cons(0,Xs)), following the left-first computation rule, is finished by one step of loop detection which unifies bit-stream(Xs) and bit-stream(cons(0,Xs)) with unifier θ={Xs/cons(0,cons(0,...))}. Note that

bit-stream(cons(0,Xs)) ← bit(0), bit-stream(Xs) (*)

is an instance of the program clause, to which we can apply unifier θ and we get another program clause instance

bit-stream(cons(0,cons(0,...))) ← bit(0), bit-stream(cons(0,cons(0,...))) (**)

We regard (*) and (**) as proof trees [4, Sec. 1.6], and notice that applying the loop detection rule extends (*) into (**), whose set I of nodes satisfies $I \subseteq T'_P(I)$, therefore I is a subset of $\mathrm{gfp}(T'_P)$. The establishment of the relation $I \subseteq T'_P(I)$ is mainly due to the reasoning that for each atom $A \in I$ (or, for each node A of proof tree (**)), there exists a program clause instance whose head is A, and whose body is a subset of I, so that if A is in I, then A is in $T'_P(I)$, indicating $I \subseteq T'_P(I)$. Such reasoning applies to all co-SLD refutation that involves use of loop detection.

[1] Logic programming works by refuting the goal, which is the negation of the proposition that is to be proven.

Definition 11 (Structural Resolution)**.** Given a logic program P and goal

$$G = \leftarrow A_1, \ldots, A_n$$

the next goal G' is derived using one of the following two rules:

1. (Rewriting Reduction) If there exists in P a clause $B_0 \leftarrow B_1, \ldots, B_m$ (with freshly renamed variables), such that $B_0 \prec_\theta A_k$ for some $k \in \{1, \ldots, n\}$, then we derive

$$G' = \leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta$$

2. (Substitution Reduction) If there exists in P a clause $B_0 \leftarrow B_1, \ldots, B_m$ (with freshly renamed variables), such that $B_0 \sim_\theta A_k$ but not $B_0 \prec_\theta A_k$ for some $k \in \{1, \ldots, n\}$, then we derive

$$G' = \leftarrow (A_1, \ldots, A_n)\theta$$

Remark. About rewriting reduction, notice that it is a special case of SLD-resolution, and since matcher θ only instantiates variables from the renamed clause $B_0 \leftarrow B_1, \ldots, B_m$ without instantiating variables from goal G, the derived goal G' can also be written as $G' = \leftarrow A_1, \ldots, A_{k-1}, (B_1, \ldots, B_m)\theta, A_{k+1}, \ldots, A_n$. About substitution reduction, notice that it is, by nature, instantiation of universal quantifier.

Definition 12 (S-Derivation/Refutation)**.** A structural resolution derivation (S-derivation for short) consists of a possibly infinite sequence $G_0, G_1, \ldots$ of goals such that for all $i \geq 0$, G_{i+1} is derived from G_i using structural resolution, without consecutive use of substitution reduction. A finite S-derivation ending with the empty goal is called an S-refutation.

Definition 13 (Computed Substitution)**.** Given a S-refutation $\mathscr{D}$, let $\theta_1, \theta_2, \ldots, \theta_n$ be the sequence of unifiers computed in $\mathscr{D}$ due to application of substitution reduction, sorted in the same order as they were computed, their composition $\theta_1\theta_2\cdots\theta_n$ is called the computed substitution from $\mathscr{D}$.

Example 13.1. Consider the program:

$$p(f(X)) \leftarrow q(X) \qquad q(a) \leftarrow \qquad r(f(a)) \leftarrow$$

Given goal $\leftarrow p(X), r(X)$ the next goal is $\leftarrow p(f(X_1)), r(f(X_1))$ by substitution reduction on $p(X)$ (with renamed clause $p(f(X_1)) \leftarrow q(X_1)$ and unifier $\theta_1 = \{X/f(X_1)\}$), then the next goal is $\leftarrow q(X_1), r(f(X_1))$ by rewriting reduction on $p(f(X_1))$ (with renamed clause $p(f(X_2)) \leftarrow q(X_2)$ and matcher $\{X_2/X_1\}$), then the next goal is $\leftarrow q(a), r(f(a))$ by substitution reduction on $q(X_1)$ (with unifier $\theta_2 = \{X_1/a\}$). Two more steps of rewriting reduction derive the empty goal $\leftarrow$ which terminates successfully the resolution and the computed substitution is the composition $\theta_1\theta_2 = \{X/f(a), X_1/a\}$.

For more details on structural resolution, see [13, 11, 15]. Next we introduce the combination of structural resolution and co-SLD style loop detection.

3 Co-inductive Structural Resolution

We introduce the declarative and operational semantics of co-inductive structural resolution. For the operational semantics we introduce how it was formulated and prove its co-inductive soundness.

3.1 Declarative Semantics

The declarative semantics of co-inductive structural resolution is chosen to be the greatest fixed point over the complete Herbrand base [17, ch. 4][23], as for co-SLD [21, 20, 3]. In fact, it was a conjecture [12] that some form of combination of structural resolution and loop detection is correct w.r.t. the greatest complete Herbrand model as co-SLD is, since they share the same co-induction mechanism and their inductive components (structural resolution and SLD resolution, respectively) are both sound and complete w.r.t. the least Herbrand model [13].

3.2 Operational Semantics

The implementation presented in Appendix A played important role in formulation of the operational semantics. The implementation was created by integrating existing implementation of structural resolution [16] and co-SLD [1], which showed plausible behaviour. So the implementation was then abstracted to obtain the operational semantics, whose soundness was later proved. The upshot is that the implementation had come before the formulation of the operational semantics, but was then verified as the soundness of the operational semantics was proved. In this section we present the operational semantics.

Definition 14 (Co-inductive Structural Resolution)**.** Given a logic program P and goal

$$G = \leftarrow (A_1, S_1), \ldots, (A_n, S_n)$$

the next goal G' can be derived by one of the following three rules:

1. (Rewriting Reduction) If there exists in P a clause $B_0 \leftarrow B_1, \ldots, B_m$ (with freshly renamed variables), such that $B_0 \prec_\theta A_k$ for some $k \in \{1, \ldots, n\}$, then let $S' = S_k \cup \{A_k\}$, we derive

$$G' = \leftarrow (A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (B_1\theta, S'), \ldots, (B_m\theta, S'), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)$$

2. (Substitution Reduction) If there exists in P a clause $B_0 \leftarrow B_1, \ldots, B_m$ (with freshly renamed variables), such that $B_0 \sim_\theta A_k$ but not $B_0 \prec_\theta A_k$ for some $k \in \{1, \ldots, n\}$, then we derive

$$G' = \leftarrow \big((A_1, S_1), \ldots, (A_n, S_n)\big)\theta$$

3. (Loop Detection) If $A_k \sim_\theta B$ for some $k \in \{1, \ldots, n\}$ and some $B \in S_k$, we derive

$$G' = \leftarrow \big((A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)\big)\theta$$

Notice that the Loop Detection rule for co-inductive structural resolution is the same as its counterpart in co-SLD, and rule-1 of co-inductive structural resolution is a special case of rule-1 of co-SLD.

Definition 15 (co-S-Derivation/Refutation)**.** A co-inductive structural resolution derivation is a possibly infinite sequence $G_0, G_1, \ldots$ where G_0 is of the form $\leftarrow (A_1, \emptyset), \ldots, (A_m, \emptyset)$ $(m \geq 0)$, and for all $i \geq 0$, G_{i+1} is derived from G_i by co-S-resolution without consecutive use of substitution reduction. A finite co-S-derivation ending with the empty goal is called a co-S-refutation.

Definition 16 (Computed Substitution)**.** Given a co-S-refutation $\mathscr{D}$, let $\theta_1, \theta_2, \ldots, \theta_n$ be the sequence of unifiers computed in $\mathscr{D}$ due to application of rule-2 or rule-3, sorted in the same order as they were computed, their composition $\theta_1\theta_2\cdots\theta_n$ is called the computed substitution from $\mathscr{D}$.

Example 16.1. Consider program:

$p(s(X)) \leftarrow q(X)$ $\quad$ $q(X) \leftarrow p(X), r(X)$ $\quad$ $r(X) \leftarrow$

In the following co-S-refutation, for each goal we always select the left most predicate to resolve.

Goal 1: $\leftarrow \big(q(X), \emptyset\big)$

Goal 2: $\leftarrow \big(p(X), \{q(X)\}\big), \big(r(X), \{q(X)\}\big)$ (rule-1. $q(X_1) \leftarrow p(X_1), r(X_1).\ \{X_1/X\}$)

Goal 3: $\leftarrow \big(p(s(X_2)), \{q(s(X_2))\}\big), \big(r(s(X_2)), \{q(s(X_2))\}\big)$ (rule-2. $p(s(X_2)) \leftarrow q(X_2).\ \theta_1 = \{X/s(X_2)\}$)

Goal 4: $\leftarrow \big(q(X_2), \{p(s(X_2)), q(s(X_2))\}\big), \big(r(s(X_2)), \{q(s(X_2))\}\big)$ (rule-1. $p(s(X_3)) \leftarrow q(X_3).\ \{X_3/X_2\}$)

Goal 5: $\leftarrow \big(r(s(s(s(\ldots)))), \{q(s(s(s(\ldots))))\}\big)$ (rule-3. $q(X_2) \sim_{\theta_2} q(s(X_2)).\ \theta_2 = \{X_2/s(s(s(\ldots)))\}$)

Goal 6: $\leftarrow$ (rule-1. $r(X_4) \leftarrow .\ \{X_4/s(s(s(\ldots)))\}$)

The answer to Goal 1 is given by computed substitution $\theta_1\theta_2 = \{X/s(s(s(\ldots))), X_2/s(s(s(\ldots)))\}$. Loop detection is used once for reduction from Goal 4 to Goal 5, and predicate $q(X_2)$ in Goal 4 is co-inductively proved.

3.3 Soundness Proof

In this section, the main result shows that given a goal G, if there is a co-S-refutation for G, with computed substitution σ, then there is a co-SLD refutation for $G\sigma$, with computed substitution ε (i.e. the empty substitution). Therefore if co-SLD is sound, then $G\sigma\varepsilon = G\sigma$ is in the greatest complete Herbrand model, meaning that co-S-resolution is also sound.

We will define a transformation algorithm that step-by-step transforms a co-S-refutation of goal G into a co-SLD refutation of goal $G\sigma$. The transformation is via an intermediate derivation, called *co-rewriting-id derivation*, which simply consists of co-SLD resolution steps interleaved with identity reduction steps (c.f. Definition 17). So given a co-S-refutation, it will be firstly transformed into a co-rewriting-id refutation, which will then be trivially transformed into a co-SLD refutation. The transformation from a co-S-refutation to a co-rewriting-id refutation is done during a sequential traverse of the co-S-refutation, starting from the initial goal. According to the three lemmas (i.e. Lemma 1, 2 and 3, defined later), each goal reduction step in the co-S-refutation establishes a co-rewriting-id reduction step, and all co-rewriting-id reduction steps established during the traverse form the co-rewriting-id refutation. The following are details of the proof.

Definition 17 (Identity Reduction)**.** Given some goal G, the reduction from G to itself, is called identity reduction, denoted by

$$G \xrightarrow{\text{id}} G$$

Definition 18 (co-Rewriting-ID Resolution)**.** Given a program and some goal G, the next goal G' can be derived from G using one of following three rules:

1. The same as rule 1 in Definition 14.
2. Identity reduction.
3. The same as rule 3 in Definition 14.

Definition 19 (co-Rewriting-ID Derivation/Refutation)**.** A co-rewriting-id derivation is a possibly infinite sequence $G_0, G_1, \ldots$ where G_0 is of the form $\leftarrow (A_1, \emptyset), \ldots, (A_m, \emptyset)$ $(m \geq 0)$, and for all $i \geq 0$, G_{i+1} is derived from G_i by co-rewriting-id resolution without consecutive use of identity reduction. A finite co-rewriting-id derivation ending with the empty goal is called a co-rewriting-id refutation.

Definition 20 (Computed Substitution)**.** Given a co-rewriting-id refutation $\mathscr{D}$, let $\theta_1, \theta_2, \ldots, \theta_n$ be the sequence of unifiers computed in $\mathscr{D}$ due to application of rule-3, sorted in the same order as they were computed, their composition $\theta_1\theta_2\cdots\theta_n$ is called the computed substitution from $\mathscr{D}$.

Proposition 1. *Given a program, for any goal G, if there is a co-rewriting-id refutation for G with computed substitution θ, then there is a co-SLD refutation for G with the same computed substitution θ.*

Proof. Suppose $\mathscr{D} = G_0, \ldots, G_n$ is a co-rewriting-id refutation for $G = G_0$ with computed substitution θ. By simultaneously removing from $\mathscr{D}$ all G_{i+1} $(i \in [0, n-1])$ such that $G_i \xrightarrow{\text{id}} G_{i+1}$, the resulting derivation $\mathscr{D}'$ constitutes a co-SLD derivation with computed substitution θ. Moreover, $\mathscr{D}'$ is a special case of co-SLD derivation since Definition 18-rule 1 is a special case of Definition 6-rule 1. □

Theorem 1. *Given a program, for any goal G, if there is a co-S-refutation for G with computed substitution σ, then there is a co-rewriting-id refutation for $G\sigma$ with computed substitution ε (the empty substitution).*

The proof of Theorem 1 is based on properties of co-inductive structural resolution rules, formulated in the following three lemmas.

Lemma 1 (Rewriting Preservation)**.** *Let*

$$G \xrightarrow[B]{\textit{rule-1}} G'$$

be a goal reduction using rule-1 (as defined in Definition 14), and B the program clause involved in the reduction.

Then for any substitution σ, it holds that

$$G\sigma \xrightarrow[B]{\textit{rule-1}} G'\sigma$$

Proof. Assume

- $G = \leftarrow (A_1, S_1), \ldots, (A_k, S_k), \ldots, (A_n, S_n)$ and
- B has the form $B_0 \leftarrow B_1, \ldots, B_m$ $(m \geq 0)$ and
- $B_0 \prec_\gamma A_k$, for some $k \in [1, n]$.

By Definition 14, rule-1,

$$G' = \leftarrow (A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (B_1\gamma, S'), \ldots, (B_m\gamma, S'), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n) \tag{1}$$

where $S' = S_k \cup \{A_k\}$.

Since $B_0 \prec_\gamma A_k$ (by the above assumption), it means (by Definition 4) that

$$B_0\gamma = A_k \tag{2}$$

Then for all σ, if we apply σ to both sides of (2), we have $B_0\gamma\sigma = A_k\sigma$, which means (by associativity of substitution [17, Sec. 4] and Definition 4) that

$$B_0 \prec_{\gamma\sigma} A_k\sigma. \tag{3}$$

Now consider $G\sigma$, by notational convention,

$$G\sigma = \leftarrow (A_1\sigma, S_1\sigma), \ldots, (A_k\sigma, S_k\sigma), \ldots (A_n\sigma, S_n\sigma) \tag{4}$$

Because of (3) and (4), we can have reduction

$$G\sigma \xrightarrow[B]{\text{rule-1}} G'' \tag{5}$$

where, by Definition 14, rule-1,

$$G'' = \leftarrow (A_1\sigma, S_1\sigma), \ldots, (A_{k-1}\sigma, S_{k-1}\sigma), (B_1\gamma\sigma, S''), \ldots, (B_m\gamma\sigma, S''), (A_{k+1}\sigma, S_{k+1}\sigma), \ldots, (A_n\sigma, S_n\sigma) \tag{6}$$

where $S'' = S_k\sigma \cup \{A_k\sigma\}$.

Compare (1) and (6), we have, by notational convention,

$$G'' = G'\sigma \tag{7}$$

By (5) and (7), we reach the conclusion of Lemma 1. □

Hereinafter we adopt the following notation for substitution compositions. Given a sequence of substitutions $\theta_1, \theta_2, \ldots, \theta_n$, for all $k \in \{1, \ldots, n\}$, let σ_k denote the composition $\theta_k\theta_{k+1}\cdots\theta_n$. For example, let $\theta_1, \theta_2, \theta_3, \theta_4$ be a sequence of 4 substitutions, then $\sigma_1 = \theta_1\theta_2\theta_3\theta_4$, $\sigma_2 = \theta_2\theta_3\theta_4$, $\sigma_3 = \theta_3\theta_4$ and $\sigma_4 = \theta_4$.

Lemma 2 (Instantiation Preservation)**.** *Let*

$$G \xrightarrow[\theta_k]{\textit{rule-2}} G'$$

be a goal reduction using rule-2 (as defined in Definition 14), where θ_k is the unifier involved in the reduction, and let

$$\sigma_k = \theta_k\theta_{k+1}\cdots\theta_n$$

for some $n > k$ and some (arbitrary and possibly ε) substitutions $\theta_{k+1}, \ldots, \theta_n$.

Then

$$G\sigma_k \xrightarrow{id} G'\sigma_{k+1}$$

Remark. The sequence of θ's in Lemma 2 will come from co-S-resolution steps when Lemma 2 is used to prove Theorem 1.

Proof. From the premise of Lemma 2, we have

$$G\theta_k = G' \tag{8}$$

Applying $\sigma_{k+1}(= \theta_{k+1}\cdots\theta_n)$ to both sides of (8) we have

$$G\theta_k\sigma_{k+1} = G'\sigma_{k+1} \tag{9}$$

Note that in (9)

$$\theta_k\sigma_{k+1} = \sigma_k$$

therefore by associativity of substitution, (9) can be written as

$$G\sigma_k = G'\sigma_{k+1}$$

hence the identity reduction $G\sigma_k \xrightarrow{\text{id}} G'\sigma_{k+1}$. □

Lemma 3 (Loop Detection Preservation)**.** *Let*

$$G \xrightarrow[\theta_k]{\textit{rule-3}} G'$$

be a goal reduction using rule-3 (as defined in Definition 14), where θ_k is the unifier involved in the reduction, and let

$$\sigma_k = \theta_k \theta_{k+1} \cdots \theta_n$$

for some $n > k$ and some (arbitrary and possibly ε) substitutions $\theta_{k+1}, \ldots, \theta_n$.

Then

$$G\sigma_k \xrightarrow[\varepsilon]{\textit{rule-3}} G'\sigma_{k+1}$$

Proof. Assume

$$G = \leftarrow (A_1, S_1), \ldots, (A_k, S_k), \ldots, (A_n, S_n) \tag{10}$$

and

$$A_k \sim_{\theta_k} B \tag{11}$$

for some $k \in [1, n]$ and some

$$B \in S_k \tag{12}$$

By Definition 14, rule-3,

$$G' = \leftarrow \big((A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)\big)\theta_k \tag{13}$$

From (11) and Definition 3,

$$A_k\theta_k = B\theta_k \tag{14}$$

Applying $\sigma_{k+1}(= \theta_{k+1} \cdots \theta_n)$ to both sides of (14), we have $A_k\theta_k\sigma_{k+1} = B\theta_k\sigma_{k+1}$, then due to $\sigma_k = \theta_k\sigma_{k+1}$ and associativity of substitution,

$$A_k\sigma_k = B\sigma_k \tag{15}$$

which means

$$A_k\sigma_k \sim_\varepsilon B\sigma_k \tag{16}$$

Consider $G\sigma_k$, which can be written as

$$G\sigma_k = \leftarrow (A_1\sigma_k, S_1\sigma_k), \ldots, (A_k\sigma_k, S_k\sigma_k), \ldots, (A_n\sigma_k, S_n\sigma_k) \tag{17}$$

Due to (12), it holds that

$$B\sigma_k \in S_K\sigma_k \tag{18}$$

From (16) and (18), $G\sigma_k$ as in (17) can be reduced using Definition 14 rule 3, resulting in

$$G'' = \leftarrow (A_1\sigma_k, S_1\sigma_k), \ldots, (A_{k-1}\sigma_k, S_{k-1}\sigma_k), (A_{k+1}\sigma_k, S_{k+1}\sigma_k), \ldots, (A_n\sigma_k, S_n\sigma_k)$$

which can be rewritten in a simpler form

$$G'' = \leftarrow \big((A_1, S_1), \ldots, (A_{k-1}, S_{k-1}), (A_{k+1}, S_{k+1}), \ldots, (A_n, S_n)\big)\sigma_k \tag{19}$$

The reduction of $G\sigma_k$ is denoted by

$$G\sigma_k \xrightarrow[\varepsilon]{\text{rule-3}} G'' \tag{20}$$

Comparing (13) with (19), we conclude, by associativity of substitution, that

$$G'' = G'\sigma_{k+1}$$

and with (20) we reach the conclusion of Lemma 3. □

Next we give an algorithm that outputs co-rewriting-id refutations, giving a co-S-refutation as input. This algorithm constitutes our proof of Theorem 1 and we will provide an example to demonstrate the algorithm at work.

Proof of Theorem 1. Given a goal $G = G_0$, assume $\mathscr{D} = G_0, \ldots, G_n$ is a co-S-derivation, and $\theta_1, \ldots \theta_m$ is the sequence of unifiers computed during derivation $\mathscr{D}$ due to the use of Definition 14 rule 2 or 3. Let $\theta_{m+1} = \varepsilon$ so $\sigma_1 = \theta_1, \ldots \theta_m\theta_{m+1}$ is the computed substitution for goal G.

Definition 14 is the default domain when we mention rule 1,2 or 3 in this proof. We build the co-rewriting-id derivation $\mathscr{D}'$ of $G\sigma_1$ using the following algorithm.

For each $i \in \{0, \ldots, n-1\}$, *starting from* $i = 0$ *and in the ascending order of* i, (*)

- If
$$G_i \xrightarrow[B]{\text{rule-1}} G_{i+1}$$
then write down, by Lemma 1,
$$G_i\sigma_x \xrightarrow[B]{\text{rule-1}} G_{i+1}\sigma_x$$
where
$$x = \begin{cases} 1 & \text{If } i = 0; \\ k & \text{If } i > 0 \text{ and } G_i\sigma_k \text{ is in } \mathscr{D}'. \end{cases}$$
x is well-defined in the second case because of the condition (*).
- If
$$G_i \xrightarrow[\theta_k]{\text{rule-2}} G_{i+1}$$
then write down, by Lemma 2,
$$G_i\sigma_k \xrightarrow{\text{id}} G_{i+1}\sigma_{k+1}$$
- If
$$G_i \xrightarrow[\theta_k]{\text{rule-3}} G_{i+1}$$
then write down, by Lemma 3,
$$G_i\sigma_k \xrightarrow[\varepsilon]{\text{rule-3}} G_{i+1}\sigma_{k+1}$$

□

Remark. Compare the specific co-S-refutation for goal G_0, which has computed substitution $\sigma_1 = \theta_1\theta_2\theta_3\theta_4$ where $\theta_4 = \varepsilon$, with the co-rewriting-id refutation for goal $G_0\sigma_1$ generated by the algorithm.

$$G_0 \xrightarrow[B_1]{rule-1} G_1 \xrightarrow[B_2]{rule-1} G_2 \xrightarrow[\theta_1]{rule-2} G_3 \xrightarrow[\theta_2]{rule-3} G_4 \xrightarrow[B_3]{rule-1} G_5 \xrightarrow[\theta_3]{rule-3} (G_6 =\!\leftarrow)$$

$$G_0\sigma_1 \xrightarrow[B_1]{rule-1} G_1\sigma_1 \xrightarrow[B_2]{rule-1} G_2\sigma_1 \xrightarrow{id} G_3\sigma_2 \xrightarrow[\varepsilon]{rule-3} G_4\sigma_3 \xrightarrow[B_3]{rule-1} G_5\sigma_3 \xrightarrow[\varepsilon]{rule-3} (G_6\sigma_4 =\!\leftarrow)$$

Theorem 2 (Soundness)**.** *Co-inductive structural resolution is sound with respect to the greatest complete Herbrand model. In other words, if a goal G has a co-S-derivation with computed substitution* σ*, then* $G\sigma$ *is in the greatest model.*

Proof. Given a program and some goal G, assume G has a co-S-derivation with computed substitution σ. By Theorem 1 $G\sigma$ has a co-rewriting-id derivation with computed substitution ε, then by Proposition 1 $G\sigma$ has a co-SLD derivation with computed substitution ε. Since co-SLD is sound with respect to the greatest complete Herbrand model, $G\sigma\varepsilon = G\sigma$ is in the model. □

4 Related Work and Conclusion

Existing soundness proof for co-SLD helped this work. A soundness proof of co-SLD, based on co-induction, is provided in [20], in which it was established by a lemma that if some goal G has co-SLD derivation with computed substitution σ, then $G\sigma$ also has a co-SLD derivation. This idea inspired the author to explore if a goal has a co-S-derivation, whether there is also a co-S-derivation for the same goal with computed answer applied. Another soundness proof of co-SLD is given in [3], which is based on the theory of infinite tree logic programming formulated in [10]. The infinite tree derivation proposed in [10] selects all sub-goal altogether rather than one sub-goal at a time, and it is sound with respect to the greatest model. In [3] the soundness of co-SLD is proved by showing that any co-SLD derivation can be unfolded into an infinite tree derivation, therefore the soundness of co-SLD derivation is backed by the soundness of infinite tree derivation. Our proof in this paper obviously is inspired by such technique in [3], which relates different derivations and reuses previous results.

As a by-product of our proof, we can use the same arguments to show that soundness and completeness of structural resolution can be proved based on soundness and completeness of SLD resolution. For soundness, if a goal G has a successful structural resolution derivation with computed substitution σ, then $G\sigma$ has a successful rewriting-id derivation, which is a special case of SLD derivation. For completeness, it needs to be shown that every SLD derivation has a corresponding structural resolution derivation, by splitting each non-rewriting step in SLD derivation into one step of substitution reduction followed by one step of rewriting reduction.

Future work will involve development of a productivity semi-decision algorithm based on co-inductive structural resolution. Now we have a sketch of the role that will be played by co-S-resolution. Given a non-terminating SLD derivation, necessarily some (maybe none) of its SLD resolution steps are rewriting reductions. A class of programs are characterized for their termination for rewriting [13, 14], called *observationally productive* programs. Since all consecutive rewriting steps are finite for such programs, in a non-terminating SLD derivation of some observationally productive program, there necessarily are infinite steps of non-rewriting SLD resolution steps. This fact will be crucial for productivity analysis since only non-rewriting steps can produce unifiers that may accumulate and instantiate the original goal into an infinite tree at infinity. Since the notion of productivity relies on rewriting reduction, productivity analysis is made easier by S-resolution compared with using SLD resolution; hence the advantage of structural resolution over SLD resolution. Moreover, loop detection needs to be combined with S-resolution to serve as a finite implementation of non-terminating productive S-derivations; finding a way for such a combination, proving its co-inductive soundness, and implementing it, are the contributions of this paper.

Acknowledgement I would like to thank my supervisor Dr. Ekaterina Komendantskaya for her support and discussion. I would like to thank Dr. Joe Wells and anonymous reviewers for their constructive comments.

References

[1] Davide Ancona (2013): *Regular corecursion in Prolog*. *Computer Languages, Systems & Structures* 39(4), pp. 142–162, doi:10.1016/j.cl.2013.05.001.

[2] Davide Ancona, Andrea Corradi, Giovanni Lagorio & Ferruccio Damiani (2010): *Abstract Compilation of Object-Oriented Languages into Coinductive CLP(X): Can Type Inference Meet Verification?* In: *Formal*

Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers, pp. 31–45, doi:10.1007/978-3-642-18070-5_3.

[3] Davide Ancona & Agostino Dovier (2015): *A Theoretical Perspective of Coinductive Logic Programming*. *Fundam. Inform.* 140(3-4), pp. 221–246, doi:10.3233/FI-2015-1252.

[4] K.L. Clark (1980): *Predicate Logic as a Computational Formalism*. Research monograph / Department of Computing, Imperial College of Science and Technology, University of London. Available at `https://www.doc.ic.ac.uk/~klc/monograph.html`.

[5] Alain Colmerauer (1985): *Prolog in 10 Figures*. *Commun. ACM* 28(12), pp. 1296–1310, doi:10.1145/214956.214958.

[6] Bruno Courcelle (1983): *Fundamental properties of infinite trees*. *Theoretical Computer Science* 25(2), pp. 95 – 169, doi:10.1016/0304-3975(83)90059-2.

[7] M.H. van Emden & J.W. Lloyd (1984): *A logical reconstruction of Prolog II*. *The Journal of Logic Programming* 1(2), pp. 143 – 149, doi:10.1016/0743-1066(84)90001-3.

[8] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara & Jan Willem Klop (2010): *Productivity of Stream Definitions*. *Theor. Comput. Sci.* 411(4-5), pp. 765–782, doi:10.1016/j.tcs.2009.10.014.

[9] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon & Ajay Mallya (2007): *Coinductive Logic Programming and Its Applications*, pp. 27–44. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-74610-2_4.

[10] Joxan Jaffar & Peter J. Stuckey (1986): *Semantics of Infinite Tree Logic Programming*. *Theor. Comput. Sci.* 46(2-3), pp. 141–158, doi:10.1016/0304-3975(86)90027-7.

[11] Patricia Johann, Ekaterina Komendantskaya & Vladimir Komendantskiy (2015): *Structural Resolution for Logic Programming*. In: *Tech. Commu. of ICLP' 15*.

[12] Ekaterina Komendantskaya (2017): *Personal communication*.

[13] Ekaterina Komendantskaya & Patricia Johann (2015): *Structural Resolution: a Framework for Coinductive Proof Search and Proof Construction in Horn Clause Logic*. *CoRR* abs/1511.07865. Available at `http://arxiv.org/abs/1511.07865`.

[14] Ekaterina Komendantskaya, Patricia Johann & Martin Schmidt (2016): *A Productivity Checker for Logic Programming*. *LOPSTR'16*. Available at `http://arxiv.org/abs/1608.04415`.

[15] Ekaterina Komendantskaya, John Power & Martin Schmidt (2016): *Coalgebraic logic programming: from Semantics to Implementation*. *Journal of Logic and Computation* 26(2), p. 745, doi:10.1093/logcom/exu026.

[16] Yue Li (2016): *Comparative Study of Search Strategies for Term-Matching and Unification Based Resolution in Prolog*. Available at `http://www.macs.hw.ac.uk/~yl55/CoALP_Report_Dec16.pdf`. Unpublished.

[17] J. W. Lloyd (1987): *Foundations of Logic Programming; (2Nd Extended Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, doi:10.1007/978-3-642-83189-8.

[18] Alberto Martelli & Ugo Montanari (1982): *An Efficient Unification Algorithm*. *ACM Trans. Program. Lang. Syst.* 4(2), pp. 258–282, doi:10.1145/357162.357169.

[19] Ben A. Sijtsma (1989): *On the Productivity of Recursive List Definitions*. *ACM Trans. Program. Lang. Syst.* 11(4), pp. 633–649, doi:10.1145/69558.69563.

[20] Luke Simon (2006): *Extending Logic Programming with Coinduction*. Ph.D. thesis, The University of Texas at Dallas. Available at `http://www.utdallas.edu/~gupta/lukethesis.pdf`.

[21] Luke Simon, Ajay Mallya, Ajay Bansal & Gopal Gupta (2006): *Coinductive Logic Programming*, pp. 330–345. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/11799573_25.

[22] M. H. Van Emden & R. A. Kowalski (1976): *The Semantics of Predicate Logic As a Programming Language*. *J. ACM* 23(4), pp. 733–742, doi:10.1145/321978.321991.

[23] W. P. Weijland (1988): *Semantics for logic programs without occur check*, pp. 710–726. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1016/0304-3975(90)90194-M.

A Implementation of Co-Inductive Structural Resolution

SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
http://www.swi-prolog.org

```
%-----------------------------------------------------------
clause_tree(true,_) :- !.
clause_tree((G,R), Hypo) :-
   !,                                                  % Note 1
   clause_tree(G, Hypo),
   clause_tree(R, Hypo).

clause_tree(A,Hypo) :- find_loop(A,Hypo).

clause_tree(A, Hypo) :-    % rewriting reduction
            unifying_and_matching_rule(A, Body),
            clause_tree(Body, [A|Hypo]).                % Note 2

clause_tree(A, Hypo) :-    % substitution reduction.
            unifying_not_matching_rule(A, _),
            clause_tree(A, Hypo).                       % Note 3

%-----------------------------------------------------------
find_loop(A,[B|_]) :- A = B.
find_loop(A,[_|C]) :- find_loop(A,C).

% choose clauses whose heads unifies with the goal,
% and specifically, matches the goal.
unifying_and_matching_rule(A, Body) :-
         copy_term(A,A_copy),                           % Note 4
         clause(A_copy,_,Ref),                          % Note 5
         clause(A1,_,Ref),                              % Note 6
         subsumes_term(A1,A),                           % Note 7
         clause(A,Body,Ref).                            % Note 8

% choose clauses whose head unifies with the goal,
% and specifically, does not match the goal.
unifying_not_matching_rule(A, Body) :-
        copy_term(A,A_copy),
        clause(A_copy,_,Ref),
        clause(A1,_,Ref),
        \+ subsumes_term(A1,A),
        clause(A,Body,Ref).
%-----------------------------------------------------------
```

Note 1 Clauses deal with mutually exclusive cases, hence the cuts.

Note 2 `A` is not instantiated by finding a matching clause.

Note 3 `A` is instantiated by finding a unifying but not matching clause.

Note 4 At run time variable `A` is bound to the current (atomic sub-)goal G, `A_copy` then is a variant $\acute{G}$ of G with fresh variables. Built-in `copy_term/2` is used to make a copy of G to use in the next procedure `clause(A_copy,_,Ref)` to search for a unifying rule without instantiating variables from G.

Note 5 At run time, this procedure finds some clause whose head unifies with the variant $\acute{G}$ of the current (atomic sub-)goal G and get the clause's reference number n that is bound to `Ref`. The term $\acute{G}$ bound to `A_copy` may be instantiated. The body of the found clause, which may be instantiated, is discarded as indicated by "_".

Note 6 Use the reference number `Ref` to get a copy of the found clause by the previous procedure '`clause(A_copy,_,Ref)`'. Only the head, which is bound to `A1`, of the clause is needed for subsumes check, and the body of the clause is discarded as shown by '_'.

Note 7 Term matching is checked by using built-in predicate `subsumes_term/2`, which does not instantiate variables. Any binding made for subsumes check will be undone by implementation of `subsumes_term/2`.

Note 8 If the subsumes check is passed by the found unifying clause, then use a fresh copy of this particular clause, as specified by `Ref`, to reduce the goal. Variables in the term t bound to `A` will not be instantiated because the clause head subsumes t as has been checked, but variables in the body of the clause, which is bound to `Body`, are instantiated by sub-terms from t.

One of the anonymous reviewers of this paper suggested that

> "The two clauses for rewriting and substitution reduction can be merged into a single one to make the interpreter more compact and efficient (but maybe a bit less readable)."

And he/she suggested the following code:

```
clause_tree(A, Hypo) :-
     copy_term(A,A_copy),
     clause(A_copy,_,Ref),
     clause(A1,_,Ref),
     subsumes_term(A1,A) *-> clause(A,Body,Ref),
                             clause_tree(Body, [A|Hypo])
                             ;
                             clause(A,Body,Ref),
                             clause_tree(A, Hypo).
```

```
% Example object programs
% ------------------------
% trace: clause_tree(a,[])
a  :- a1,a2.
a1 :- b1,b2.
b1 :- c1,c2.
a2.
b2.
c1.
```

```
c2.
%-------------------------
% trace: clause_tree(p(X),[]).
p(s(X)) :- p(X),p(X).  % non-linear co-recursion
%-------------------------
% trace: clause_tree(cond_f(X),[]).
cond_c(s(a)).
cond_e(s(_)).
cond_f(X) :- cond_e(X),cond_c(X).
%-------------------------
% trace: clause_tree(q(X),[]).
r(_).
p(s(X)) :- q(X).
q(X) :- p(X),r(X).
```